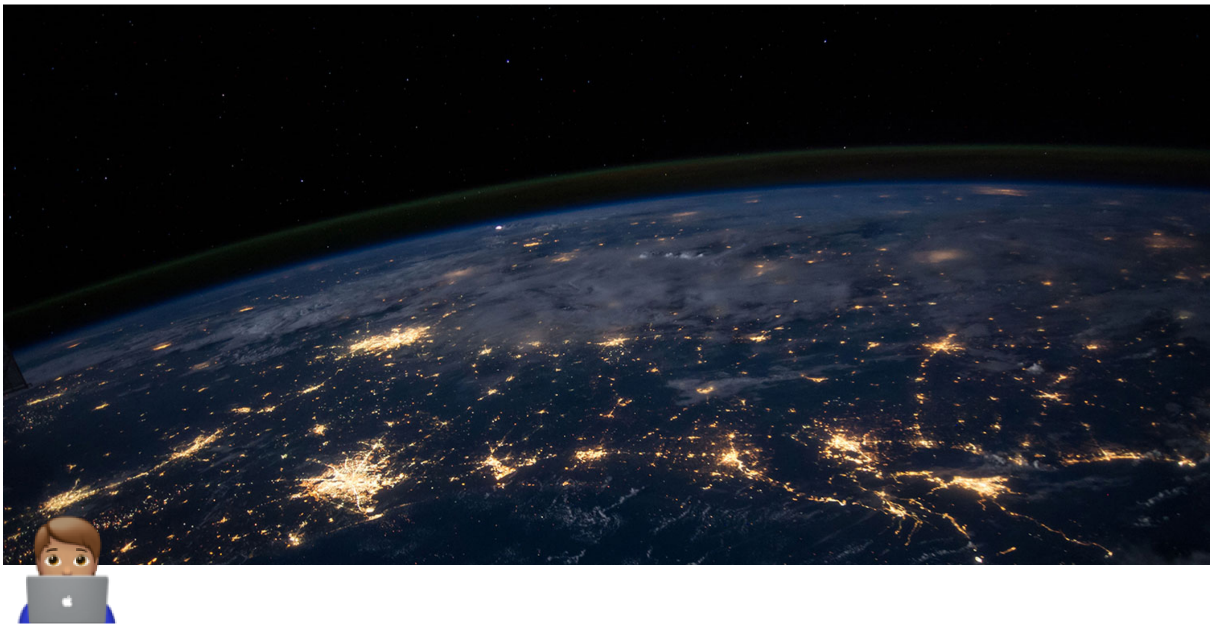


Security Vulnerabilities in Code

- **What are Security Vulnerabilities?**
    - Security vulnerabilities are weaknesses in a system or application that attackers can exploit to compromise confidentiality, integrity, or availability. Understanding common vulnerabilities is key in building more secure applications, particularly in a DevSecOps pipeline.
- **Common Vulnerabilities**:
    - SQL Injection
    - Cross-Site Scripting (XSS)
    - Cross-Site Request Forgery (CSRF)
    - Insecure Deserialisation
    - Improper Input Validation

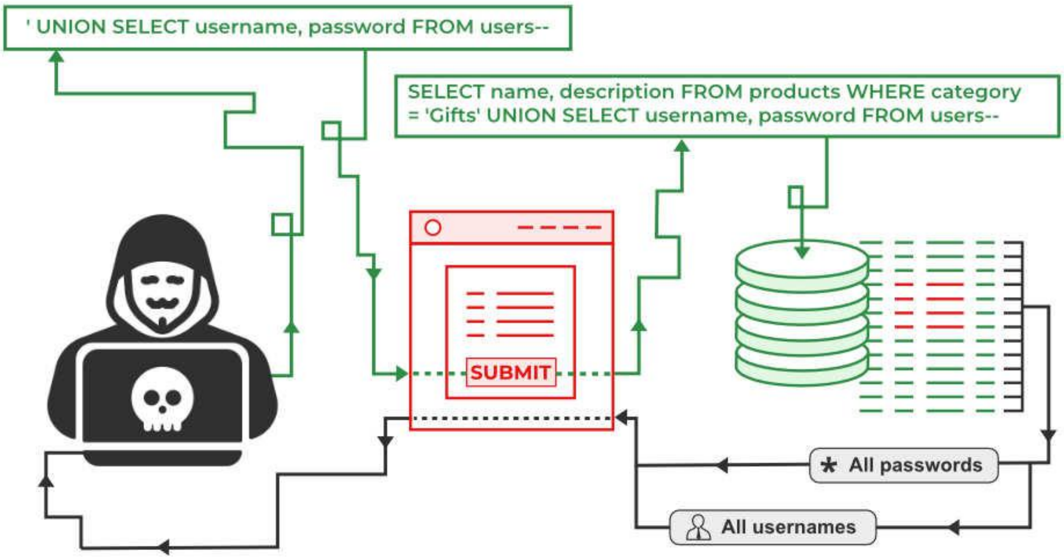
▼ SQL Injection

What is it?

- SQL Injection is a technique where attackers manipulate an application's SQL queries by injecting malicious SQL code through user inputs (e.g., forms or URL parameters).

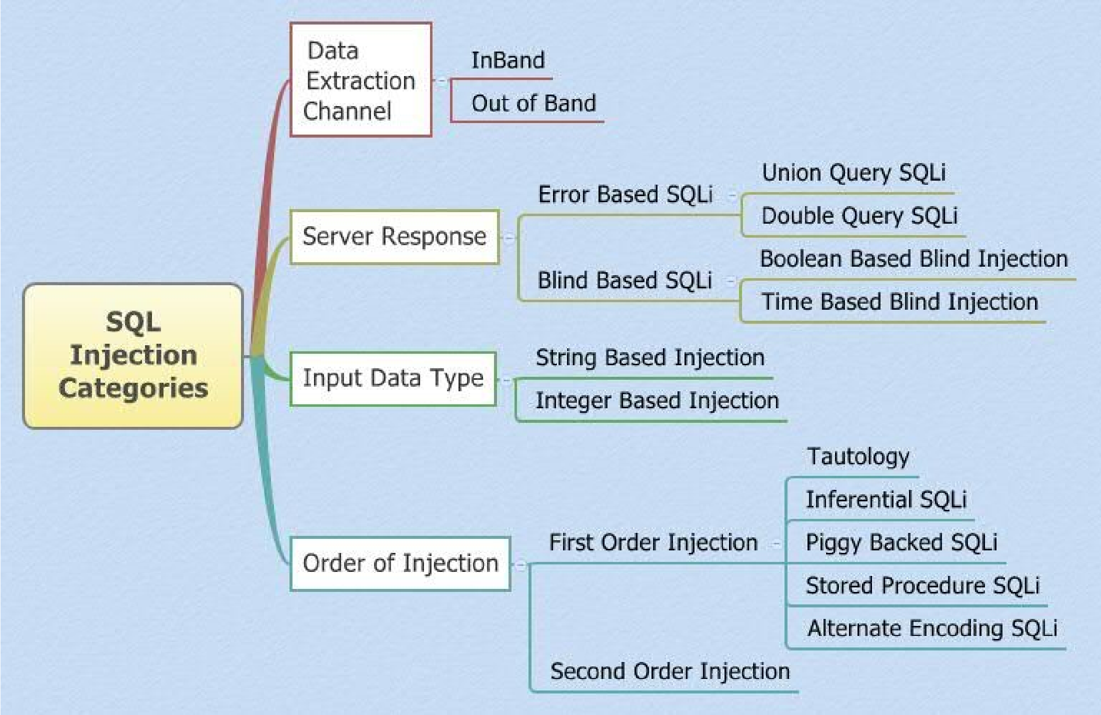
- If a user input is not properly sanitised, the attacker can execute arbitrary queries to retrieve, modify, or delete sensitive data.
- **Example**:
    - Query: `SELECT * FROM users WHERE username = 'user' AND password = 'password';`
    - Attack: `SELECT * FROM users WHERE username = 'user' AND password = ''; OR '1'='1';`

  Result: Attacker logs in without providing a valid password.


' UNION SELECT username, password FROM users--
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT username, password FROM users--
SUBMIT
All passwords
All usernames

▼ Types of SQL Injection

- **Classic SQL Injection**: Occurs by inserting SQL queries in user inputs.
- **Blind SQL Injection**: SQL queries are injected without knowing the output.
- **Error-based SQL Injection**: Attacker retrieves information from error messages returned by the database.



Demo: `hackapp`

- **Initial version** of the `loginUser` method used direct string comparison, leading to SQL injection vulnerability.
- Code snippet:

```
@PostMapping("/api/login")
public String loginUser(@RequestParam String username, @RequestParam String password) {
    User user = userRepository.findByUsername(username);
    if (user != null) {
        return "Welcome, " + user.getUsername();
    }
    return "Invalid username or password";
}
```

- The method does not protect against SQL injection.

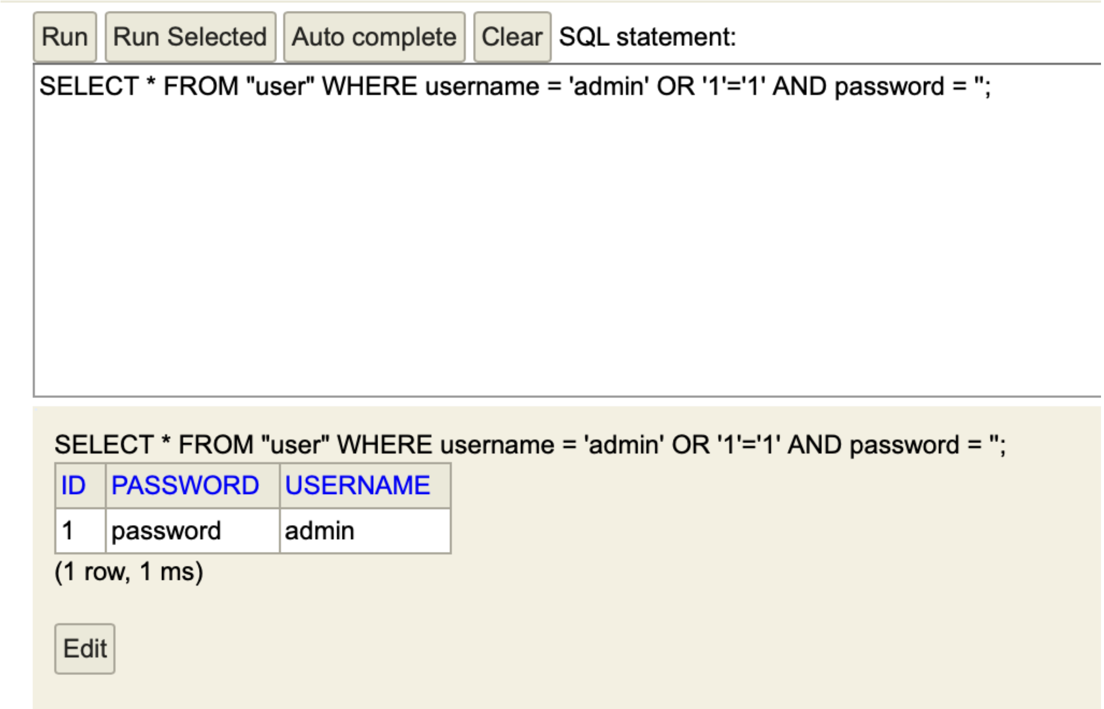
Understanding SQL Injection

- If the `findByUsername` method directly queries the database without validation, attackers can manipulate SQL.

- **Example** SQL query with injected data:

  ```
  SELECT * FROM "user" WHERE username = 'admin' OR
  '1'='1' AND password = '';
  ```

- **Impact**: The SQL injection bypasses authentication and allows unauthorized access.



```
SELECT * FROM "user" WHERE username = 'admin' OR '1'='1' AND password = '';
```

SELECT * FROM "user" WHERE username = 'admin' OR '1'='1' AND password = '';

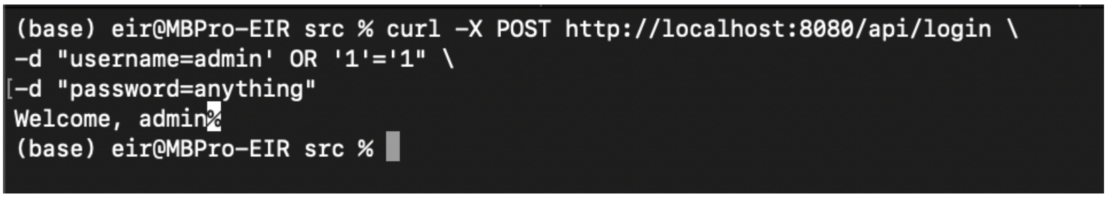
ID	PASSWORD	USERNAME
1	password	admin

(1 row, 1 ms)

SQL Injection on `hackapp` via `curl`

- We've implemented a simple vulnerable login system in `hackapp` for demonstrating SQL injection.
- Use the following `curl` command to simulate an SQL injection attack:

  ```
  curl -X POST http://localhost:8080/api/login \
  -d "username=admin' OR '1'='1" \
  -d "password=anything"
  ```



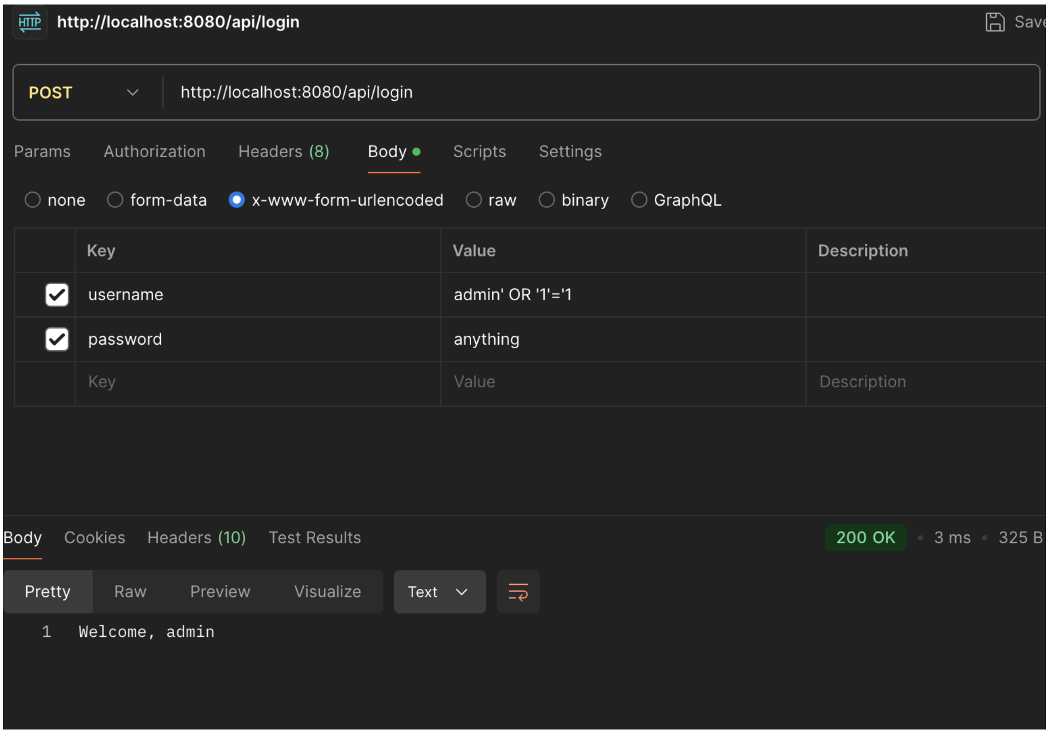
```
(base) eir@MBPro-EIR src % curl -X POST http://localhost:8080/api/login \
-d "username=admin' OR '1'='1" \
[-d "password=anything"
Welcome, admin%
(base) eir@MBPro-EIR src %
```

- **Why This Works**:
    - In our JDBC-based implementation, the SQL query dynamically includes the input values without proper sanitation, making it susceptible to SQL injection.
    - This bypasses the login mechanism by altering the SQL query structure.

▼ SQL Injection on `hackapp` via `Postman`

- **Step-by-Step Postman Demonstration**:
    1. **Open Postman** and create a new request.
    2. Set the request method to **POST**.
    3. Enter the following URL in the address field:

```
http://localhost:8080/api/login
```



POST http://localhost:8080/api/login

○ none ○ form-data ● x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

	Key	Value	Description
☑	username	admin' OR '1'='1	
☑	password	anything	
	Key	Value	Description

200 OK • 3 ms • 325 B

```
Welcome, admin
```

4. Under the **Body** tab, select **x-www-form-urlencoded** and add the following key-value pairs:
    - **Key**: `username`

      Value: `admin' OR '1'='1`

    - **Key**: `password`

      Value: `anything`

  5. Click **Send**.

- **Expected Result**:
  - The SQL injection works, and you should be logged in as the first user (`admin`) in the database, bypassing the password check.
  - Postman will display a response indicating a successful login.

Preventing SQL Injection

- Instead of manually writing SQL, **use JPA's** `findByUsernameAndPassword` to securely query the database.
- **Code**:
  - The repository method uses **parameterised queries** with Spring Data JPA to avoid SQL injection:

```
User findByUsernameAndPassword(String username, String password);
```

```
@PostMapping("/login")
public String loginUser(@RequestParam String username, @RequestParam String password) {
    User user = userRepository.findByUsernameAndPassword(username, password);
    if (user != null) {
        return "Welcome, " + user.getUsername();
    }
    return "Invalid username or password";
}
```

  - JPA handles argument sanitization, preventing SQL injection by default.
  - Use prepared statements or parameterized queries to prevent SQL injection.

```
@Query("SELECT u FROM User u WHERE u.username = :username")
User findByUsername(@Param("username") String username);
```

How to Prevent SQL Injection?

- **Input Validation**: Ensure all user inputs are properly validated.
- **Parameterised Queries** (Prepared Statements): Use placeholders for query parameters to prevent direct user input into SQL queries.
- **Stored Procedures**: Use database-stored procedures instead of dynamic SQL queries.
- **ORM Frameworks**: Use ORM (Object-Relational Mapping) tools such as Hibernate, JPA, etc., which generate secure queries.
- **Least Privilege**: Ensure the database user has minimal privileges.

Other Common Vulnerabilities

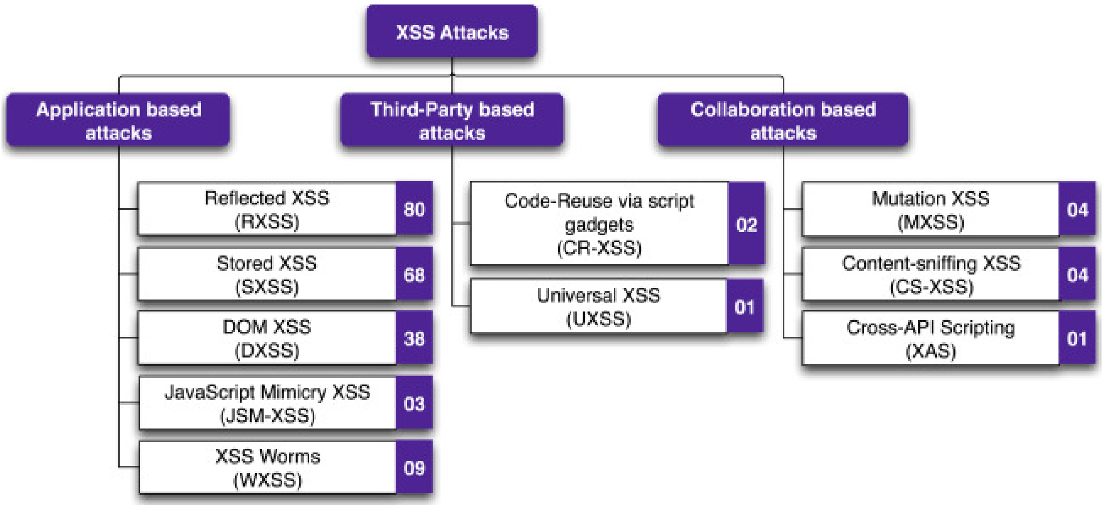
- **Cross-Site Scripting (XSS)**:
  - Definition: Injecting malicious scripts into trusted websites viewed by other users.
  - Prevention: Input sanitization, encoding user input.
- **Cross-Site Request Forgery (CSRF)**:
  - Definition: Forcing a logged-in user to perform unwanted actions on a website.
  - Prevention: Use tokens (e.g., CSRF tokens) to verify requests.
- **Insecure Deserialisation**:
  - Definition: Deserialisation of untrusted data that can lead to remote code execution.
  - Prevention: Validate and sanitise serialised data.

Cross-Site Scripting (XSS)

- **What is XSS?**

  - A type of injection attack where malicious scripts are injected into otherwise benign and trusted websites.
  - The attacker tricks the victim's browser into executing malicious scripts, potentially compromising their sensitive information (like cookies or login tokens).

- **Types of XSS:**


XSS Attacks
Application based attacks
Third-Party based attacks
Collaboration based attacks
Reflected XSS (RXSS) 80
Stored XSS (SXSS) 68
DOM XSS (DXSS) 38
JavaScript Mimicry XSS (JSM-XSS) 03
XSS Worms (WXSS) 09
Code-Reuse via script gadgets (CR-XSS) 02
Universal XSS (UXSS) 01
Mutation XSS (MXSS) 04
Content-sniffing XSS (CS-XSS) 04
Cross-API Scripting (XAS) 01

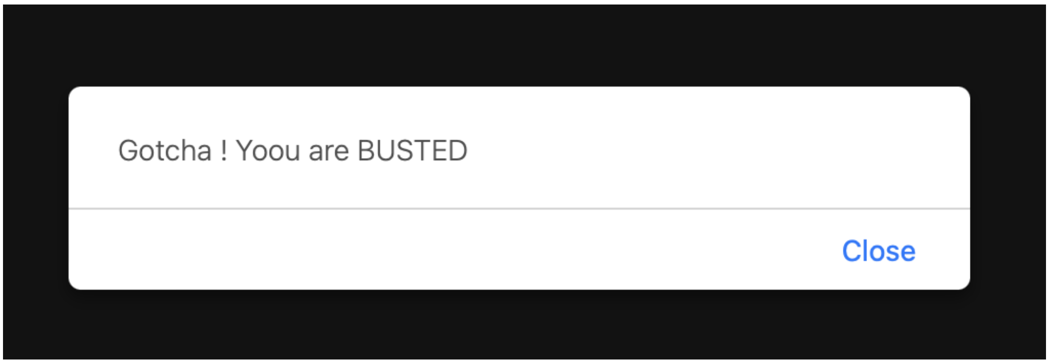
  1. **Stored XSS**:
     - Malicious script is **permanently stored** on a target server (e.g., in a database).
     - When users visit the page, the malicious script is delivered to their browser from the server.
  2. **Reflected XSS**:
     - The injected script is reflected off a web server, often via an error message or search result.
     - The user is tricked into clicking a malicious link, where the script is injected and executed.
  3. **DOM-based XSS**:
     - The vulnerability occurs in the browser rather than the server. The malicious script is part of the **Document Object Model**.

▼ XSS on `hackApp` via `http`

- **Steps**:

1. Open your browser.
2. Use the following URL:

```
http://localhost:8080/api/profile?username=<script
>alert('Gotcha ! You are BUSTED');</script>
```



Gotcha ! Yoou are BUSTED

3. **Expected Result**: A JavaScript `alert` will pop up displaying "`Gotcha ! You are BUSTED`"

▼ XSS on `hackApp` via `Postman` (POST Request)

1. **Create a New Request** in Postman:
   - **Method**: Select `POST`.
   - **URL**: Enter the URL for your `ProfileController`:

     ```
     http://localhost:8080/api/profile
     ```

2. **Add the Malicious Payload**:
   - Under the **Body** tab in Postman:
     - Select `x-www-form-urlencoded`
     - In the key-value fields:
       - Key: `username`
       - Value: `<script>alert('Gotcha ! You are BUSTED');</script>`
3. **Send the Request**:
   - Click on the **Send** button.
4. **Observe the Response**:

- The response should return HTML where the `username` parameter is reflected in the response body, unsanitised.
- Postman will return the following response:

  ```
  <html><body><h1>Profile Updated: <script>alert('Gotcha ! You are BUSTED');</script></h1></body></html>
  ```

- If you open this response in a browser, it should trigger an alert showing the "`Gotcha ! You are BUSTED`" message, indicating a successful XSS attack.

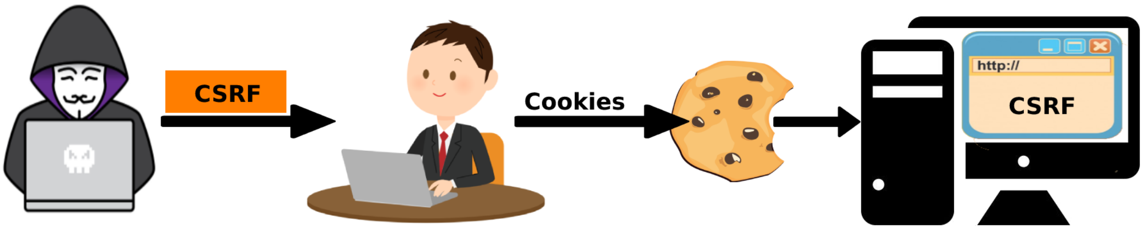
Why XSS is Dangerous

- **Impact of XSS**:
  - **Data Theft**: Attacker can steal cookies, local storage data, and other sensitive info.
  - **Session Hijacking**: Attacker can impersonate users by stealing session tokens.
  - **Defacement**: Altering the content of the web page for all users.
- **Real-World Example**:
  - In 2019, an XSS vulnerability in a popular CMS allowed attackers to inject malicious scripts into administrator panels.

Mitigating XSS

- **Input Validation**: Always validate and sanitize user inputs.
- **Escape User Output**: Escape any user input before rendering it in the HTML (e.g., `<`, `>`, `&`, `"`).
- **Content Security Policy (CSP)**: Use a strict CSP to block inline scripts and only allow trusted scripts to be executed.
- **HTTPOnly Cookies**: Prevent JavaScript from accessing cookies.

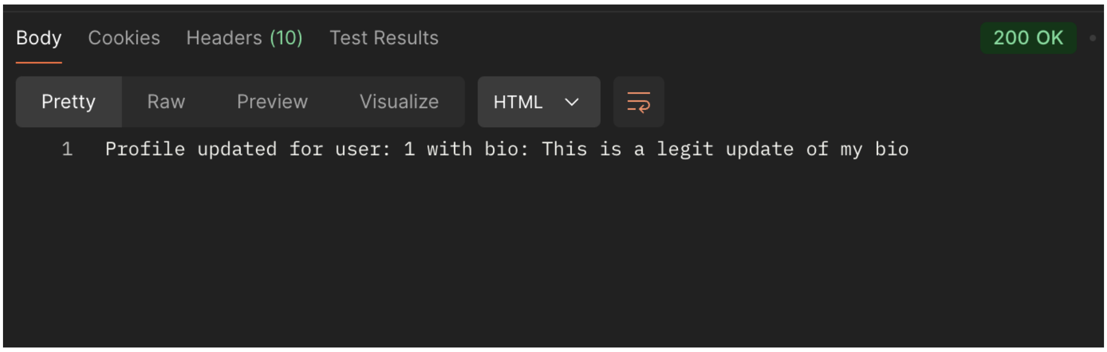
Cross-Site Request Forgery (CSRF)



- **What is CSRF?**
  - An attack where a user is tricked into performing actions they didn't intend to, by sending an unauthorised request to a trusted web application.
  - This usually happens when the user is already authenticated and their browser automatically includes credentials (e.g., cookies, tokens).
- **CSRF Example**:
  - A malicious website contains a form that sends a request to a bank's API to transfer money.
  - The user is already logged into their bank account. When they visit the malicious site, the browser unknowingly sends the request with the user's bank credentials.

▼ Legitimate Request

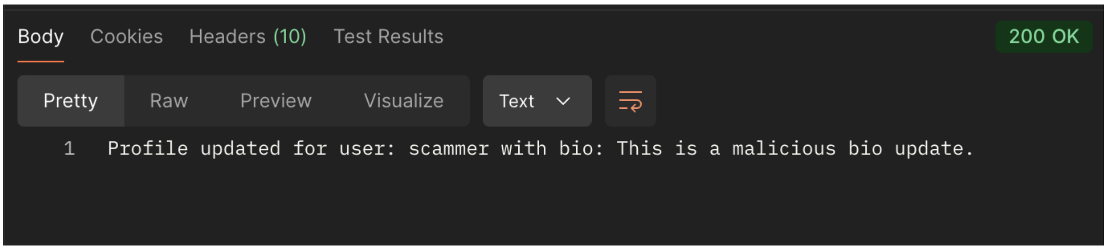
- Use Postman to make a legitimate profile update for user with `userId=1`.
- **URL**: `http://localhost:8080/api/updateProfile`
- **Method**: `POST`
- **Body**:
  - `username=1`
  - `bio=This is a legitimate bio update.`
- **Expected Result**:



```
Profile updated for user: 1 with bio: This is a legit update of my bio
```

CSRF Attack on `hackApp` via Postman

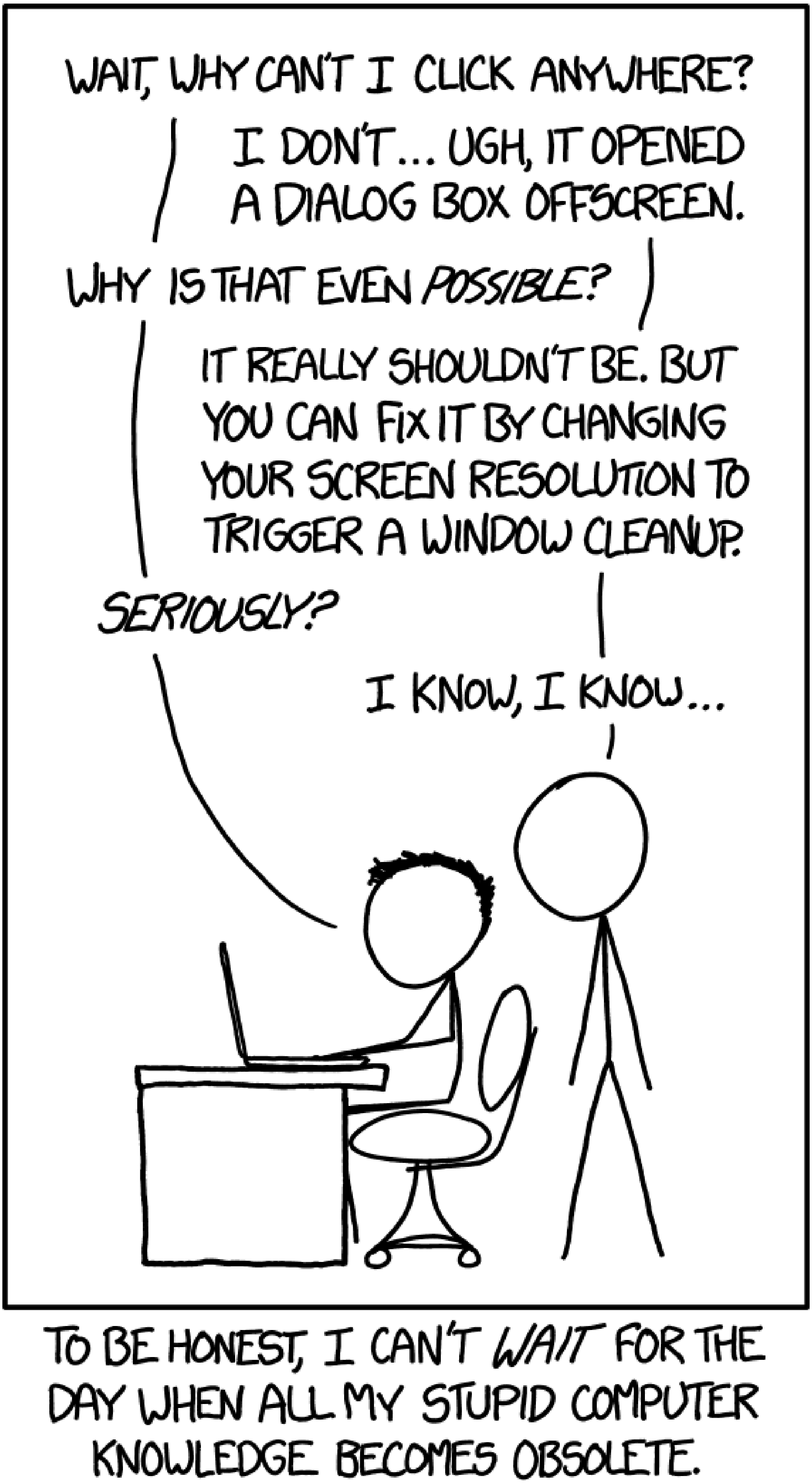
- Use Postman to simulate a malicious request.
- **URL**: `http://localhost:8080/api/updateProfile`
- **Method**: `POST`
- **Body**:
    - `username=scammer`
    - `bio=This is a malicious bio update.`
- **Expected Result**:



```
Profile updated for user: scammer with bio: This is a malicious bio update.
```

Mitigating CSRF Attacks

- **How to Prevent CSRF**:
    - Use CSRF tokens for all state-changing requests.
    - Validate the origin or referer headers to ensure the request comes from trusted sources.
    - Ensure that critical actions (e.g., updating profile data) are protected by verifying the authenticity of the user session.


WAIT, WHY CAN'T I CLICK ANYWHERE?
I DON'T... UGH, IT OPENED A DIALOG BOX OFFSCREEN.
WHY IS THAT EVEN *POSSIBLE*?
IT REALLY SHOULDN'T BE. BUT YOU CAN FIX IT BY CHANGING YOUR SCREEN RESOLUTION TO TRIGGER A WINDOW CLEANUP.
SERIOUSLY?
I KNOW, I KNOW...
TO BE HONEST, I CAN'T *WAIT* FOR THE DAY WHEN ALL MY STUPID COMPUTER KNOWLEDGE BECOMES OBSOLETE.